**University of Alberta**

# AN EXERCISE IN USING SCL TO CAPTURE BEHAVIORAL DESIGN INTENTIONS IN A WEB APPLICATION

by

**Anjan Sen**

A thesis submitted to the Faculty of Graduate Studies and Research in partial fulfillment of the requirements for the degree of **Master of Science**.

Department of Computing Science

Edmonton, Alberta

Fall 2007

Library and Archives Canada

Published Heritage Branch

395 Wellington Street
Ottawa ON K1A 0N4
Canada

Bibliothèque et Archives Canada

Direction du Patrimoine de l'édition

395, rue Wellington
Ottawa ON K1A 0N4
Canada

*Your file Votre référence*
*ISBN: 978-0-494-33346-4*
*Our file Notre référence*
*ISBN: 978-0-494-33346-4*

NOTICE:
The author has granted a non-exclusive license allowing Library and Archives Canada to reproduce, publish, archive, preserve, conserve, communicate to the public by telecommunication or on the Internet, loan, distribute and sell theses worldwide, for commercial or non-commercial purposes, in microform, paper, electronic and/or any other formats.

The author retains copyright ownership and moral rights in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

AVIS:
L'auteur a accordé une licence non exclusive permettant à la Bibliothèque et Archives Canada de reproduire, publier, archiver, sauvegarder, conserver, transmettre au public par télécommunication ou par l'Internet, prêter, distribuer et vendre des thèses partout dans le monde, à des fins commerciales ou autres, sur support microforme, papier, électronique et/ou autres formats.

L'auteur conserve la propriété du droit d'auteur et des droits moraux qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

---

In compliance with the Canadian Privacy Act some supporting forms may have been removed from this thesis.

While these forms may be included in the document page count, their removal does not represent any loss of content from the thesis.

Conformément à la loi canadienne sur la protection de la vie privée, quelques formulaires secondaires ont été enlevés de cette thèse.

Bien que ces formulaires aient inclus dans la pagination, il n'y aura aucun contenu manquant.

Canada

*I think there is a world market for maybe five computers.*

– Thomas J. Watson, IBM Chairman, 1943.

*To My Mother,*

*Who brought me to this world.*

# Abstract

Many errors in systems are caused by mismatch between the intentions of the designers of the system and the actions of the implementers of the system, even when they are the same people. This thesis examines how structural aspects of a system expressed in SCL (Structural Constraint Language) can be used to capture behavioral intentions in a simple web application.

# Table of Contents


**1 Introduction** 1

**2 Static and Dynamic Analysis** 3

**3 Related Work** 5

**4 The E-Voting System** 11

4.1 Business Objects . . . . . 11

4.2 Implementation Issues . . . . . 13

**5 Introduction to SCL** 14

**6 Some Security Questions** 22

6.1 Enforcing Architecture . . . . . 23

6.2 Enforcing Workflow . . . . . 24

6.3 Enforcing Complex Properties . . . . . 27

**7 Qualitative Experience** 36

**8 Comparison with other testing tool** 39

**9 Conclusion** 42

**10 Acknowledgments** 44

# List of Figures


4.1 Class Diagram of the Online Voting System . . . . . 12

5.1 Abstract syntax of SCL . . . . . 15

5.2 Basic types and the subtype relation . . . . . 15

6.1 Administrator Login Page . . . . . 25

6.2 New Voter Registration . . . . . 26

6.3 Sequence diagram of one-voter-one-vote scenario . . . . . 28

6.4 Voter Login . . . . . 34

# Chapter 1

# Introduction

A web application or webapp is an application that is accessed via world wide web through a network such as internet. There is huge popularity of webapps due to the ability of these applications to be updated and maintained without distributing new softwares to thousands of users distributed over the globe. In our everyday life we use lots of variations of web applications like webmail, online shopping, wikipedia, discussion board, weblog, electronic voting and many others. In a typical scenario, web applications dynamically generate a series of web documents in some standard format supported by common browsers. While each individual web page is delivered to the user as a static document, the sequence of web pages can provide the user with an interactive experience. While there are several probable variations of web applications, almost all of them contain some generic issues like form processing, navigation through sequence of web pages, back-end connection with database, authentication of users, handling of errors generated during execution flow etc. These are primarily behavior issues. The focus of this thesis is how well we can use static analysis of structure to ensure proper behavior. We chose online voting or e-voting as an example application in our investigation because it provides a domain that is simple enough to implement, yet has sufficiently real behavior. In the next few paragraphs we try to point out the specific requirements and issues of a typical online voting system that we developed.

In every election process great care is taken to ensure the free, fair and smooth running of the whole vote casting and counting process. Key aspects of any election include issues such as administrative control, integrity and availability of the

system, ensuring authenticity of voters but anonymity of votes, and the correct tabulation of the votes. These issues can all be lumped under the fuzzy notion of *security*. The idea of conducting elections using commodity hardware and software over publicly available networks, so called *e-voting*, raises many security issues.

Technology has always been associated with security, whether it is using ballots to replace the counting of hands, or the internet to facilitate voting over distance in space and time. Each technology brings advantages and problems. A particular technology will have an intended use, that is, the users of the technology have an idea of how it is to be applied to address a particular problem. It will also have an actual deployment. Problems in a system arise because the technology is not deployed as intended (an implementation error), or that the deployment admits use beyond what the designers intended (an under-specification error). Processes to evaluate the security of a system (or its correctness in general) are designed to check that intentions are properly handled throughout the life cycle of the system.

Our focus here is on static analysis of source code, and the particular tool we describe is called SCL (Structural Constraint Language) [4]. SCL can be used to capture and confirm that the intended properties of the voting system are respected during the implementation of the system. We hope that the reader of this thesis should develop some intuition on how to design behavioral features of an online system in a structured way. And the typical security related questions of e-voting provide us with the opportunity to examine the behavioral features within a manageable level of domain.

Like most powerful tools, SCL is non-trivial to learn and employ. To appreciate the strengths and weaknesses of static analysis tools like SCL we need to go into the gory details of its use. From our experience with different online projects we have deduced that it is generally difficult to capture all the underlying aspects in a general domain, There are many potential security intentions of every online application and trying to apply SCL on all or most of them is not feasible for our current project. Therefore, we have decided to investigate an online voting system which definitely limits the domain to a more specific area of online applications.

# Chapter 2

# Static and Dynamic Analysis

There are two types of program analysis: static and dynamic. To understand the capabilities of different security analysis tools to we need to understand the difference between the two approaches. The majority of the web and network application systems use weak versions of both: dynamic in the form of testing, static in the form of type-checking. However, both forms have their pros and cons [2]. Both techniques can produce false positives: detecting an error when none exists; and false negatives: missing an error that is present!

Static analysis tools parse and analyze the source code without running it. Compiler optimization and type checking are standard static analyses. Since the static analysis tools do not execute the source code, they typically perform a conservative analysis in an attempt to reduce false negatives. In this case conservative approach simply means an overestimate of the program behavior that is guaranteed to predict all of the behavior of the program we are interested in. So their results need to be generalized across any number of executions of program or tool. This imposes a restriction on the expressiveness of the static tools and causes them to produce false positives in the output. It might also introduce false negatives as they have to approximate program behavior to run faster.

On the other hand, dynamic analysis tools, which generally talk about behavior, execute the program at real time and try to gauge the behavior of the code over a number of runs. Unit testing is an standard dynamic analysis tool. There is no need to abstract or approximate the program behavior as we have the full knowledge of the program paths followed by the tool and the data used to do the testing. Dynamic

tools are typically as fast as the actual program execution. However, we might face the problem of generalization of the input data and the execution paths followed during the analysis.

To understand the effect of a conservative approach and the notion of false positive and negative we can think about an example. Lets assume that we are trying to create a static buffer overrun detection system in a high-level programming language like C. In this case a false negative is defined as a real buffer overrun that our detection system could not find, whereas false positive is the scenario where our system mistakenly flagged a buffer overrun. As it is computationally intractable to statically detect buffer overruns, we might need to make a conservative assumption that there will be very low occurrence of false negative; which means we make an assumption that the scenario of our detection system missing an actual buffer overflow is very unlikely to happen.

Both forms of analysis have their usefulness. Each form of analysis observes a different set of program executions, limited by approximations (static analysis) or by the input set (dynamic analysis.) Dynamic analysis tools have to worry more about false negatives, while static analysis tools have to worry more about false positives. Dynamic analysis is fast, while static analysis is slow. However, it's often easier to introduce static analysis early in the development cycle. Some organizations introduce a fast static analysis as an automatic step in compilation or source code checking, while using a longer, more in depth dynamic analysis for security reviews. Having said that the line between static and dynamic analysis is fuzzy. There are tools which cannot be unequivocally classified as static or dynamic. For example model checking is not about running the program, rather it is about running the approximation of the program. As a general rule, we can debate that static analysis is more concerned about the structure of the code, while dynamic analysis tries to find the behavior of the code.

# Chapter 3

# Related Work

Web applications have a widespread use in the real world, which makes it convenient to discuss their structure. Based on the structure, content, layout and logic behind development we can classify typical web applications into two classes: static and dynamic. In the static structure all parts of a web application are stored and administered in a publishing server. Triggered by an explicit release or time-based mechanism the publishing server generates the static web pages only once and transfer them to the web. From there they are accessible to the users. As the web page generation is done only once, this approach reduces the server's load, though this has very limited usefulness due to the lack of dynamic activity from the users. On the other hand, when following dynamic structure, web pages are generated on the fly based on specific user requests to the server of the system. When a request is sent to the web server from the web browser, the server retrieves the necessary data from the database (or the file system) and generates the output media, e.g. an HTML page. In a complex system we might have several languages embedded into the HTML page. Dynamic web pages usually consist of some HTML code and a dynamic part, which is code written in another language (may be JSP or Javascript) that generates HTML. The code that generates HTML can do this based on variables in a template or on code. The text to be generated may come from a back-end database. Because of the presence of multiple code bases in a web application, we might need to deal with them structurally, by having well-defined coding conventions.

In any web application the primary intention of the developers is to ensure the

security of the system as a whole. And any source code written to implement a web application plays a major part in analyzing the behavioral design intentions of the system. Even after installing state of the art firewalls, guarding against off-the-shelf software patches and protecting the system with heavy encryption, there may be many loopholes in the structure to attack the system. We shall try to discuss briefly different possibilities of attacks on a web application.

Some very common types of attacks on a web application are SQL injection, cross-site scripting, user authorization etc. SQL injection technique exploits a security vulnerability of the database of a web application when user input from the front-end is either incorrectly filtered for string escape characters of SQL statements or the input is not strongly typed. Basically everything doable through SQL on a database can be done through SQL injection like fetching, modifying and deleting information. Another very worrying problem for developers of web application is cross-site scripting (XSS). In this problem, the target of attack is actually the browser of the client side of the application. XSS happens when a web site allows input from one user to be displayed in the browsers of other users without being properly filtered. An included javascript may get access to cookies of a client machine and thus leak the session Id and other personal information of that user to the intruder. Another type of problem might occur with web application which deals with user authorization, where authorization is done to check if an user has access to a particular portion of the system. In a typical web application we might have various types of user inputs taken at different stages which might reference parts of the system that have access restrictions depending on the type of the user. The programmer has to check against the possibility of the incoming data controlled by an attacker.

Any web application environment contains many of the same development and operation challenges that we might encounter in a typical cross-platform, distributed system. As the execution of a web application is split across multiple user environments, including but not limited to uncontrolled client-side and third-party systems, we always have to cope with a lack of visibility into the end-to-end behavior of the program. This causes significant challenges in building and maintaining a reliable

web application used by a large and variable user base. As a simple example of the variable nature of the typical user systems, sending an XML-RPC request requires calling an ActiveX object in IE6, but a native JavaScript object in Firefox. However the difficulties caused by the variable nature of behavior across different systems can be overcome by better web service management and careful software design [6].

Though it is a common practice in software world to apply some kind of formal analysis like model checking, it has been realized that applying model checking to the verification of a web application is not always straightforward [1]. Most of the web applications are full of much complexity in terms of the programming languages used to develop them and the variation in the client-side system; which makes is rather difficult to extract models from such an application. As the application of model checking on a web based system is no more easier than actually coding the system, this analysis technique is not used regularly in web applications.

In a typical web application we may need to apply testing in different areas of the whole system. Starting from the simplest classes, the developers will need to program test cases to ensure that even the smallest units of the application behave correctly. Potentially each component can pass the unit tests alone - however the developers need to make sure that all the different parts work together; in other word developers might need to perform some sort of integration testing of the system. In case of web applications, HttpUnit, a test framework based on JUnit, is widely used as it allows automated test script implementation during testing cycles. But in many software projects, developers might use some sort of ad-hoc approach to cut down the development and testing time [14]. To meet the short time framework, developers might need to consider some trade off between the implementation of business logic and the quality of the product. In case of a web application expecting wide variety of user base, it is highly imperative for the developers to clarify the basic requirements of the users. As well as fulfilling these requirements, developers also have to assess the performance of the system, heavy network loads and different kinds of clients (hardware, OS, browsers).

When we focus our attention to an specific type of web application like e-voting,

it is found that significant work is done in discussing the behavior issues related to the development of such a system. Mercuri et.al. [11] presented a detailed list of features that should be present in an electronic voting system to make it acceptable under specific conditions. Some major features listed there are:

[i] separation of voter identity from vote information

[ii] recording and tabulation of vote information

[iii] audit trail of vote record

[iv] provision to produce vote confirmation to voters

[v] authentication of personnel concerned with election process

[vi] security of the whole system from outside influence

We find great insight into e-voting system by going through Neumann's [12] thorough study of key security considerations, such as system integrity, accountability, availability, and reliability, voter authenticity and data confidentiality etc. They [12] also mentioned that to be treated as acceptable any voting system must also conform with whatever election laws may be applicable for the election under concideration. We also have to ensure that the voting system is run independent with any other application running concurrently and the ballot images are stored appropriately in-case of post-election result challenge by any concerned person or group. Having said that, we also have to consider the fact that there are always some criteria elements present in the voting system that are inherently unsatisfiable under the acceptable time and money constraint.

Similar analyses are presented in [9] and [13]. Lauer et.al. [9] presented a comparative study between two different e-voting systems, regular internet voting and direct recording electronic (DRE) voting. They presented the result of a survey carried on both security experts and non-experts about the possible problems in carrying out electronic voting. Whereas the experts' concerns are system and programming errors followed closely by attempts to hack the system to alter the election result, the non-experts are mainly concerned with low voter turnout due to public distrust of e-voting systems. We find a rather pessimistic view about the future of e-voting in Rubin's paper [13]. They concluded that even though there is a certain amount of fraud existing in the current offline voting system, we tend to tolerate this

as there is no better alternative. The localized nature of the offline system makes it highly unlikely to propagate the effect of fraud to influence the outcome of an election beyond a limited area. But in case of electronic voting the possibility of fraud is higher, specially if the exact same system is deployed over a whole region. Given the present state of high availability of computers in peoples' homes, the vulnerability of the internet due to denial of service attack and the unreliability of the domain name service, it is concluded in [13] that the technology is not reliable enough to enable remote electronic voting in public elections.

A security analysis of the source code of a paperless electronic voting system is provided by Kohno et.al. [7]. They performed an analysis of the April 2002 snapshot of Diebold's AccuVote-TS 4.3.1 electronic voting system using publicly available source code. Their analysis detected several problems including unauthorized privilege escalation, incorrect use of cryptography, network vulnerabilities and poor software development process. They proposed the use of voter-verified audit trail that allows an electronic voting system to produce a paper trail of the votes for the voters to check and verify before casting their vote. Their argument in favor of this proposal is that having installed such a trail in the system, the correctness burden on the voting terminal's code should be significantly lowered. This paper trail can also be used to resolve any dispute in the final result.

There is a similar type of electronic voting system review carried out by University of California researchers on several state of California e-voting systems [15]. They conducted security audits of three e-voting systems: Sequoia, Diebold, and Hart. Serious physical and technical security vulnerabilities were found with all the three systems. In the Sequoia system, the review team found loopholes that enabled them to access the system that calculates the checksum of the data stored in the disk. They also found problems with Microsoft SQL server 2000 used to store the data. With the Diebold system, the researchers found significant discrepancies between the configurations of the machines provided by the vendors and the official description of the configuration. This particular system was also using a version of Windows 2000 server that can be easily manipulated by experienced programmers from outside. The review team concluded that the security mechanism provided by

the different e-voting systems were inadequate for public use of these systems.

Keller [5] describes important privacy considerations for a prototype voting system that includes an open source, PC-based voting machine that prints a voter-verified paper ballot along with an electronic audit trail. They discussed different privacy issues inherent in e-voting system. Their discussion amplifies the importance of careful and thorough planning during system design and implementation in order to satisfy the delicate privacy concerns. Changes in various aspects of a voting system, like voter signing-on, navigation across the ballot, checking, changing and casting vote and the applicability of a paper trail are discussed by Herrnson et.al. [3].

While all the previous papers are concerned with common voting areas like authentication, vote casting, ballot storage and result publication, Laskowski et.al. [8] investigated a rather less discussed aspect of the system - wording and placement of instructions on ballot paper. Their study suggests that improper wording and placement of instructions may cause serious implications on voters while casting their vote. That is, a mismatch between intended and actual use creates a security issue.

Having gone through various discussions and proposals about electronic voting system, it can be deduced that the current state of the technology is not reliable enough to replace the offline voting system with a full-fledged e-voting system. There are plenty of issues concerned with security, availability and acceptance of the electronic system which need to be answered properly within the limit of acceptance to the general public before we can apply e-voting at a large scale.

# Chapter 4

# The E-Voting System

Our toy system is a prototypical web-delivered voting application. We make the architectural assumption that users will have HTML/JavaScript compliant browsers on arbitrary machines connecting over an open network. We assume, unrealistically, that the underlying hardware and software of users has not been compromised. We are only going to focus on the web services aspects of the voting system: the server side code and browser side HTML/JavaScript.

During the implementation, we have designed the whole system from ground up, asking ourselves what we intend to represent as security aspects of the system. We have come up with some potential security issues which, to us, are vital for the correct and smooth running of electronic voting. The next sections are used to describe the high-level design of our system. Later we shall come up with some potential security questions and describe how SCL may be applied to address those.

We begin with a design of the business objects in the system, which then enables us to develop a high-level overview of the different voting processes it has to support. This will constitute the informal intentions of our design. To apply SCL, we then need to capture aspects of our intentions that can be expressed in terms of the source code.

## 4.1 Business Objects

In the following discussion, we refer to the whole election system as *Election*, the personnel conducting the system as *Administrator*, all the electorate as *Voter*, all

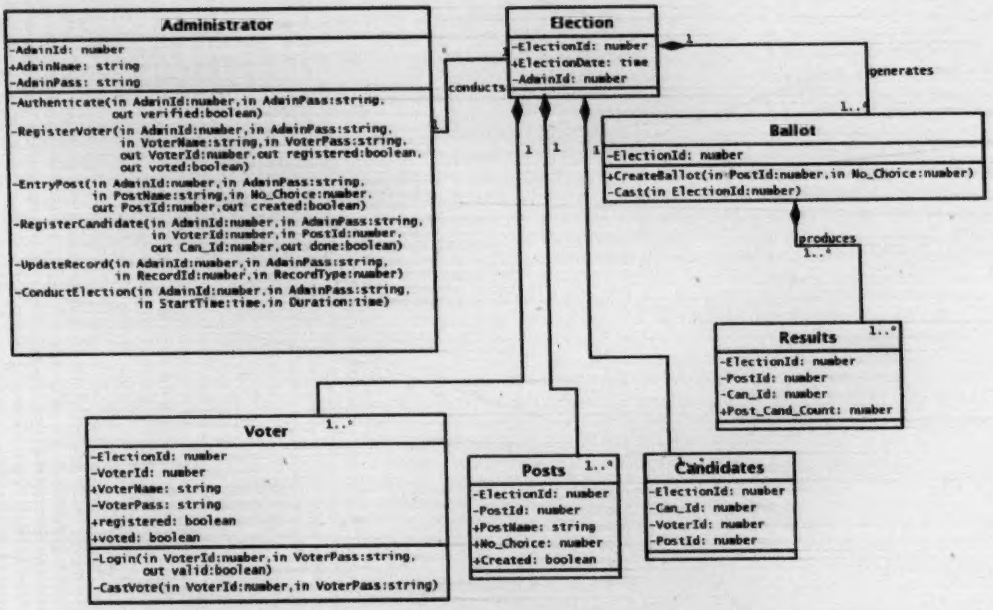


Figure 4.1: Class Diagram of the Online Voting System

the positions canvassed for as *Posts* (also commonly called positions or offices), all the people seeking election as *Candidates* and the web page containing all the options (about posts and candidates) as *Ballot*. The final outcome of polling will be presented in a separate web page called *Results*. In Fig. 4.1 we present a high-level class diagram of the whole election system under consideration.

Figure 4.1 shows the relationships among the different business objects from the client point of view. The *composition* relationship (drawn with filled diamond) between *Voter* and *Election* represents the idea that there will be no existence of *Voter* if there is no *Election*. The same can be said for the relationships between *Posts-Election*, *Candidates-Election* and *Results-Ballot*. Also, every *Election* system will include at least one or more *Voter*, *Posts*, and *Candidates*. The one-to-one relationship between *Administrator* and *Election* represents the idea that all the personnel involved with a particular election conduction duties are considered as one entity.

In our application, most of the tasks associated with election are handled under administrative control. These include registration of voters from a pool of eligible

persons, creation of posts to be canvassed in the election, registration of eligible voters as candidates for created posts etc. Once a voting system is made available online, the eligible, registered voters will be allowed to log into the system and cast their votes. After casting of votes and gathering voter confirmation about their choices, the ballot information in stored in the database without any direct connection to the voter identity. At the termination of voting period, all the collected ballot information is processed to generate the final result.

## 4.2 Implementation Issues

The e-voting system consists of a web application server that voters connect to. On the server-side we used the Tomcat http server, with Java Server Page (JSP) technology to generate web pages dynamically. All data is stored in the open source database MySQL. All programming was done in the Eclipse framework. The SCL processor plugs into Eclipse.

We have to make the pragmatic assumption that all these components actually work together, so that we can focus on the security aspects of our own code. But even under this assumption, a simple application is quite complex as it is in a multi language environment involving languages like Java, JSP, JavaScript, HTML and SQL. This poses additional challenges to reasoning about security, as implementation details are diffused over many parts of the system.

Unfortunately this current version of SCL only handles Java; so we can only address Java code that we have written, or that has been automatically generated through JSP. Obvious extensions to SCL could handle SQL queries and other static artifacts. In principal, SCL is not tied to any specific programming language. This makes it possible to apply the same SCL rules on voting systems developed in another language, say C++. In our conclusion we will speculate on how SCL could be applied to multi-linguistic applications.

The next section is dedicated for a brief introduction of SCL as our project drew heavily on this constraint language. A full introduction is in [4].

# Chapter 5

# Introduction to SCL

SCL [4] was developed to capture structural relationships between the components of an object-oriented program. It consists of a specification language for describing static relationships (or constraints) between code elements, and a checker that verifies that code satisfies the relationships. The checker is integrated into the development environment so that the developer gets immediate feedback about rule violations.

SCL notations are based on first-order logic containing sets and sequence operations. The term language of SCL consists of a set of functions reflecting the entity-relationships in the graph representations of object-oriented programs. We present a brief introduction of SCL language based on the abstract syntax of Fig. 5.1. All of the figures and tables presented in this chapter are taken from [4].

At the highest level, SCL specifications consist of sequences of interleaved declarations and formulas. An SCL constraint is a combination of a top-level formula and all the declarations that it refers to. The value of a variable in SCL is defined within a certain scope. Each declaration binds a variable to an associated expression within that scope. Logical formulas are special types of expressions yielding values of boolean type. Therefore, SCL allows us to define boolean variables with corresponding formulas as their value expressions.

We can introduce local variables for expressions through a syntactic structure called a block and there can be at most one associated block for each expression. Local variables can be used to avoid long or repeated expressions.

To ensure well-defined truth value, every SCL specification is strongly typed.

```
SCL_spec = Stmt*
Stmt = Decl | Form
Decl = ['def'] Var 'as' Expr

Form = '!' Form| Form '&' Form| Form '|' Form|
Form'=>'Form |Form'<=>'Form| Ex | Univ | Expr
Ex = 'exist' BVar_Decl+ 'hold' Form
Univ = 'for' BVar_Decl+ 'hold' Form
Expr = Var | Const | Op | Expr_With_Vars
BVar_Decl = Var':' Expr
Expr_With_Vars = '['Decl+']' Expr

Op = Set_op | Seq_op | Rel | SCL_fct
Set_op = Set_compr | Set_enum | member | cardinality|...
Set_compr = '{' Expr '|' BVar_Decl+ Form '}'
Set_enum = '{' Expr* '}'
Seq_op = ith (seq, index) | indexOf (ele, seq) |...
Rel = > | >= | < | <= | =
SCL_fct = Str '(' Expr* ')'
Const = 'true' | 'false' | Quoted-string | Integer |'packages' ...
```

Figure 5.1: Abstract syntax of SCL

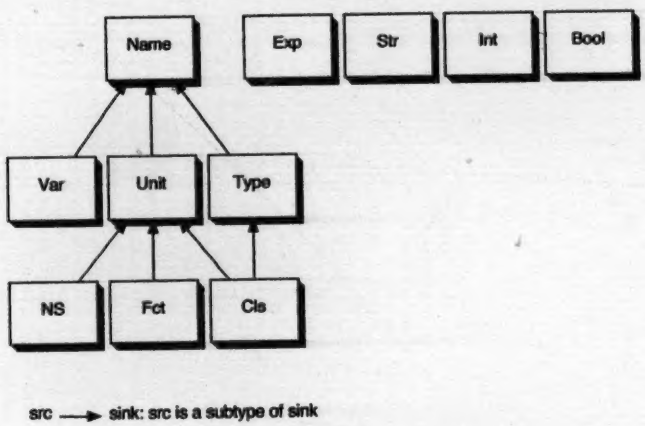


Figure 5.2: Basic types and the subtype relation

Table 5.1: SCL constructors

| Operation | Description |
|---|---|
| class: **Unit×Str→Cls** | Constructors are used to specify a program entity. The first arguments specify the program unit (packages, classes, and functions) where the entity is defined, followed by details such as names of the entity. For example, the parameter -less method m of class C can be specified as method(class(global,"C"),"m"). Note that global can be omitted. |
| var: **Unit×Str→Var** | |
| function (method): **Unit×Str×Type*→ Fct** | |

The fundamental types of SCL are basic (Fig. 5.2) and compound types. Basic types include *Str* for string values, *Int* for integers, *Bool* for boolean values, *Exp* for expressions, *Name* for named entities, *Var* for variables, *Unit* for program units that organize source code, *Type* for types, *NS* for name spaces, *Fct* for functions and *Cls* for classes. Compound types are for sets or sequence of entities , such as set of classes, or a sequence of parameters.

SCL asserts properties of a program by viewing it as a structure. Such assertions are defined based on generic constructs from first-order logic and a rich set of total functions. We present some of the constructs and functions in the tables collected from [4]. To represent different program entities, SCL uses some constructs shown in Table 5.1. At the global level, program units like packages, classes and functions are used to organize the program. Table 5.2 contains functions to examine the containment relation formed by these program units.Another important aspect of program is its type information which is shown in Table 5.3. Table 5.4 lists the operations on expressions.

From Table 5.4, we are going to discuss about one particular function ***conds*** which is used extensively in our project. According to SCL definition of [4], ***conds***. ***conds: Exp1 → Set Exp2*** returns the set of expressions (***Exp2***) that this expression (***Exp1***) control-depends on transitively. To simplify this definition we can say that the existence of expression ***Exp1*** is dependent upon the existence of expression ***Exp2***. There is a similar type of control-dependency function written as ***cd:***

Table 5.2: SCL scope operations

| Operation | Description |
|---|---|
| classes: **Unit → Set Cls** | Returns the set of classes defined within a Unit. |
| exprs: **Unit → Set Exp** | Returns the set of expressions defined within a Unit. |
| vars: **Unit → Set Var** | Returns the set of variables defined within a Unit. |
| functions (methods): **Cls → Set Fct** | Returns the set of functions defined within a Cls. |
| params: **Fct → Seq Var** | Returns the sequence of parameters of Fct. |

Table 5.3: SCL type operations

| Operation | Description |
|---|---|
| subclasses: **Cls → Set Cls** | Returns the set of subclasses of the argument class. |
| descendants: **Cls → Set Cls** | Returns the set of descendant classes of the argument class. |
| type: **Exp → Type** | Returns the static type of a given expression. |
| type: **Var → Type** | Returns the type of a given variable. |
| returnType: **Fct → Type** | Returns the return type of a given function. |
| class: **Type → Cls** | Casts a type into a class. returns "undefined" if fails. |
| isArray: **Type → Bool** | Returns true if Type is an array type. |

Table 5.4: SCL expression operations

| Operation | Description |
|---|---|
| receiver (primary): **Exp → Exp** | Returns the receiver of the argument. Returns "undefined" if no receiver. |
| args: **Exp → Seq Exp** | Returns the sequence of arguments of an expression, including the receiver. |
| cd: **Exp → Set Exp** | Returns the set of expressions that transitively control-depend on this expression. |
| conds: **Exp → Set Exp** | Returns the set of expressions that this expression control-depends on transitively. |
| dep: **Exp×Exp → Bool** | Returns true if the value of the first expression depends on that of the second. |
| uses: **Exp → Set Exp** | uses(e) returns the set of expressions that contain either expression e or a variable aliased to e. |
| function (method): **Exp → Fct** | Returns the function that a given expression is statically bound to. |
| var: **Exp → Var** | Returns the variable that a given expression is statically bound to. Returns "undefined" if not a variable. |
| refd: **Exp → Name** | If the argument is a reference expression, returns the referred entity. Otherwise, "undefined". |
| literalType: **Exp → Type** | If the argument is a reference to a type name, returns the type. Otherwise, "undefined". |
| int: **Exp → Int** | Returns the integer value if the expression is an integer constant, otherwise, "undefined". |
| isLiteralNull: **Exp → Bool** | Returns true if the expression is the null pointer (0 for C++). |

***Exp1* → *Set Exp2*** where the existence of expression ***Exp2*** is dependent upon the existence of expression ***Exp1***.

In the next paragraph we are going to present a simple example of SCL usage on Java language. It illustrates how structure can be used to give some insight into behavior.

Our example is concerned about the application of 2-D array. Let us check the following Java code (*ArrayTest.java* in *com* package):

```
1  public class Arraytest {
2  public static void main(String[] args) {
3  int a[][]= {{1,2,3},{4,5}};
4  int i,j,k=0;
5  for(i=0;i<a.length;i++)
6  for(j=0;j<a[i].length;j++) (L1)
7  //for(j=0;j<a.length;j++) (L2)
8  k = k+a[i][j];
9  System.out.println(k);
10  }
11  }
```

Going through this small Java code snippet, it is evident that to correctly access the second dimension of a 2-D array, we should use ***L1*** rather than ***L2***. But if we do use ***L2*** and try to execute the code, there will not be any error message from the compiler; as there is no syntax error with ***L2***. But if the programmer intention is to calculate the correct summation of all the values in a 2-D array, we have to use ***L1***. SCL can help us in this type of situation by determining whether or not the appropriate indexes are used to access an array. We have to check that during the second array access through the *for* loop we are reusing the first index (in this case *i*). We use the *lt [less than]* operator of SCL to find out the loop. A probable SCL rule looks like this:

*SCL Rule : Check Illegal Array Access*

```
1  def c as class("com.Arraytest")
2
3  for m:  methods(c), e:  exprs(m) holds
4  [def firstindex as var(ith(args(e), 0)); (P1)
5  def firstlength as ith(args(e),1);
6  def firstarray as var(ith(args(firstlength), 0))]
7  (
8  method(e)=lt & isDefined(firstarray) & method(firstlength)=fieldaccess
```

```
9  =>
10 ex e1:  cd(e) holds // there must exist j < a[i].length (P2)
11 [def secondindex as var(ith(args(e1),0)); (P3)
12 def secondrhs as ith(args(e1),1);
13 def lhsofsecondrhs as ith(args(secondrhs),0);
14 def secondarray as var(ith(args(lhsofsecondrhs),0));
15 def repeatindex as var(ith(args(lhsofsecondrhs),1))]
16 (
17 method(e1)=lt
18 & method(lhsofsecondrhs)=arrayaccess (P4)
19 & method(secondrhs)=fieldaccess
20 & secondarray = firstarray
21 )
22 &
23 for e1:  cd(e) holds // for all j < a[i].length (P5)
24 [def secondindex as var(ith(args(e1),0)); (P6)
25 def secondrhs as ith(args(e1),1);
26 def lhsofsecondrhs as ith(args(secondrhs),0);
27 def secondarray as var(ith(args(lhsofsecondrhs),0));
28 def repeatindex as var(ith(args(lhsofsecondrhs),1))]
29 (
30 method(e1)=lt
31 & method(lhsofsecondrhs)=arrayaccess (P7)
32 & method(secondrhs)=fieldaccess
33 & secondarray = firstarray
34 =>
35 repeatindex =firstindex
36 )
37 ) otherwise error(<<e>>, "subscripts error !!!!")
```

In this rule, the first part (***P1***) is used to define the array and the index to access the first dimension of this array. After that we move to ***P2*** where we check the existence of an expression with two indices (*i* and *j*), where *j* is the second index different from the first one *(i)*. ***P3*** defines the second index and separates the different part of the expression checked in ***P2***. In ***P4*** we make sure that the same array name is used for both the array access operation, because otherwise all the checking will make no sense. Parts from ***P2*** to ***P4*** are checking the existence of the second array access. After that we have to check that for any such array access we actually do reuse the first index. ***P6*** does the same thing as ***P3***. Then we move to ***P7*** where the reuse of first index in the second array access is verified. As pointed out before, without the reuse of index the complier will not issue any error message as syntactically we do not have to reuse the index. But to correctly calculate the summation

of a 2-D array (which should ideally be the intention of the programmer) we need to ensure that we reuse the first index.

As shown in this example SCL is useful to investigate issues which might otherwise be overlooked and generate incorrect result. Having looked into the SCL language from a very high-level of discussion we can now concentrate on how this may come handy in an online voting system.

# Chapter 6

# Some Security Questions

The seemingly straightforward online voting system is riddled with many security issues. The supervisor handles the tasks of registration of voters and candidates and creation of posts. So we have to make sure that the identity of the supervisor is properly validated. Once that is done, we have to ensure that all the necessary preconditions are satisfied while performing the other operations. For example, no person should be allowed to register as a voter more than once. The same is true when the supervisor tries to create a new post. Before any person is registered as a candidate, it should be checked whether that person is an eligible voter or not. There are issues in the vote casting phase also. We have to check the identity of every person trying to log in as a potential voter besides making sure that no person is allowed to vote more than once and no eligible voter is deprived of the right to cast his/her vote. While storing ballot information, necessary precautions must be taken to ensure the anonymity of the voter. Since a full security audit is beyond the scope of this project, we selected three problems that represent different kinds of questions one might ask.

1. Enforcing architecture through checking against illegal class access in code,
2. Enforcing workflow by ensuring the correct order of class access
3. Enforcing complex properties by checking that all the necessary preconditions are verified before an object is created within code

Before going into the details of theses security questions we want to point out the procedures that are followed during our system development. While we were

working on the system we always had to think about the structure and workflow of the source code because the specific type of SCL rules that we want to use have a very close relationship with the program. We need to think about what functions or procedures we are going to use in the program that will represent the programmer intents depicted by the design of the system. At the same time we need to maintain a very close eye on how to actually test these representations of intentions through SCL. SCL checking is too narrow in the sense that it has to strictly follow the exact design flows of the program. In other words, what can be tested using SCL is dependent too much on the design of the system. Even if we alter the design of a portion of the system without altering its behavior or outcome, we may need to change the structure of SCL tests that we created earlier for that specific part of the system. Because of this restriction indirectly imposed on our system by SCL tool, we had to come up with the SCL rules at the same time we program the system. During the development phase our first tactic was to come up with a design to represent the flow of execution for a certain function that we want to implement. After that we write the program needed for correct implementation according to our design; besides writing the SCL testing codes. If any change in the design of a part of the system is needed later we try to modify the corresponding part of the program and the SCL code side by side.

## 6.1 Enforcing Architecture

The architecture of an application has a significant impact on security. Good design can ensure that some security properties are immediate consequences of the architecture. For example, if two classes cannot communicate directly, then any information flow between them has to be via an intermediary class or external data structure. As a general rule, the less communication between classes the easier it is to deal with security. We want to restrict certain classes from talking to each other at all - for example the Candidate class should not have access to the Administrator class. Other communication should always have to pass through a gatekeeper class. SCL rules that apply globally to the application can be used to enforce architectural

decisions.

*[SCR1] SCL Rule : Check Illegal Class Access*

*Intent:* This rule that says that the only method calls that a business class other than *Election* can make are to methods in *Election*.

*Rule:*

```
1  def p as package("election.jsp");
2  def controller as class(p, "Administrator");
3  def servants as { x | x:classes(p) x != controller}
4  for c:  servants [
5  def others as { s | s:servants s != c } ]
6  m:  methods(c), e:  exprs(m) (
7  method(e) => !  in(e, methods(others))
8  );
```

*Evaluation:* This rule begins by defining *p* as the package containing all the election business classes, and identifies *controller* as the class with responsibility for controlling the application. The *servants* are defined as all the other business classes. The rule says that no servant class should ever call another servant class. The rule is read as, for every servant class *c*, define *others* as the other servant classes, then for every method *m* in *c* and for every expression *e* in of *m* if the expression is a method then it must not be in the possible methods of another servant.

To simplify our exposition, *Ballot* and *Results* are not permitted to communicate, we do allow servants to talk to the controller, and we allow backdoor communication via classes that are not business classes in the voting application.

## 6.2 Enforcing Workflow

The next case we consider relates to workflow. We want to ensure that only an authenticated administrator can access the admin functions page. If we assume that only a page redirect can cause the admin functions page to be generated, then we would like to say that every admin page redirect is guarded by an authentication.

The typical login page takes the admin user name and password and checks them against the database. The Java code snippet of Code 1 is employed to do that. Here we are checking that the submitted admin username-password combination is valid. Then we proceed to the voter registration page whose interface looks like Fig. 6.2.

Code 1. Administrator Login Code

```
1
```

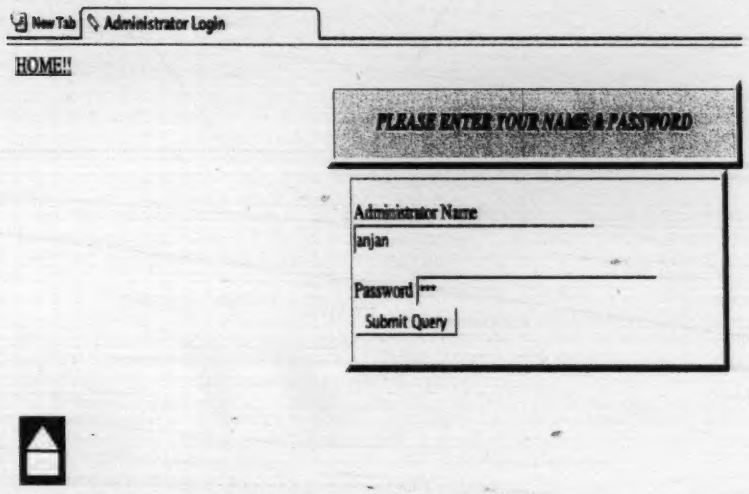


Figure 6.1: Administrator Login Page

```
2  if (request.getParameter("SUBMITTED") != null) {
3  Admin a = Admin.findAdmin(request.getParameter ("adname"),
 request.getParameter("adpass"));
4  if (a == null) {
5  error = "No such Admin found, try again..";
6  }
7  else {
8  sendRedirect("adminfunction.jsp");
9  }
10  }
```

What we intend in this rule is that no page ever generates a redirect to the admin functions page unless the redirect is guarded by a successful admin login. Note, this rule implies that the administrator has to re-authenticate every time they leave the admin page.

This is a case where the structure of the JSP code implies a structure on the web pages, which implies a structure on the navigation, which then implies restrictions on the workflow. This subtle chain of dependencies is a weakness of using structure to enforce behaviour. An SCL rule like the following can be applied to enforce the dependency of administrator function page being loaded under the control of *findAdmin* function of *Administrator* class:

*[SCR2]*

*Intent:* SCL Rule to Check Conditional Dependency

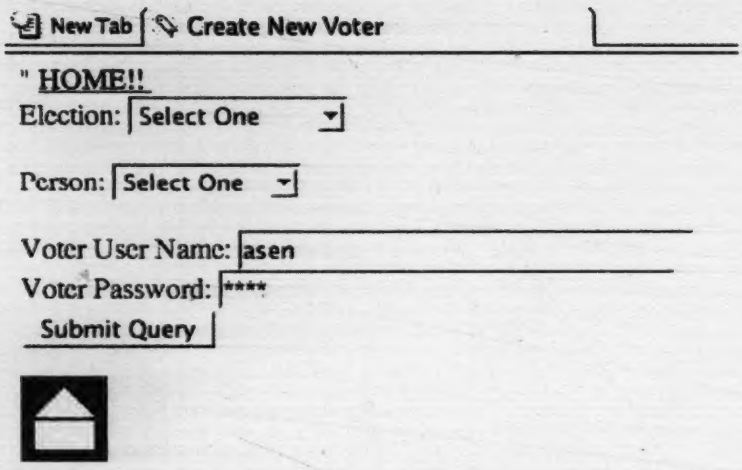


Figure 6.2: New Voter Registration

*Rule:*

```
1  e:  exprs(Administrator) holds
2  ( (name(method(e))="sendRedirect("adminfunction.jsp")")
3  =>
4  ex ce:  conds(e) holds ( name(method(ce))="Admin.findAdmin"
   )
5  )
```

*Evaluation:* This rule checks to make sure that the execution of *sendRedirect* function comes under the control of *findAdmin* function of *Administrator* class. One weakness of this rule is that it can only check the presence of *sendRedirect* under *findAdmin*, but it cannot check the presence of any parameter required by either of these two functions.

The login process in Code 1 is typical of JSP web forms. A web page is first fetched by the client browser through a GET http request. When the user selects a control on the form (usually a submit button of some flavour), the page data is sent back to the server via a http POST request. In the JSP world, the incoming request is stored in the request object, and the various fields of submitted form are accessed through the getParameter method. The intended structure of the code for a form is that of a fetch and inspection of a form field associated with a control

(SUBMITTED in this case), followed by fetching and acting on the values of fields in the form. These typical patterns can be encoded in SCL, and can be used to detect omissions from form handling code.

## 6.3 Enforcing Complex Properties

To discuss this property, we investigate the 'One-Voter-One-Vote' rule.

In our proposed voting system, registration of eligible voters is performed under administrative control. There is an externally supplied list of all the eligible persons and the voters are selected from that list. Every voter is assigned a unique combination of user name and password for a particular election to ensure that no voter is registered more than once. Having said that, the same person may be registered as an eligible voter for more than one election. During the vote-casting phase, registered voters log into the system and cast their votes. We can divide this scenario into several steps for thorough discussion.

Steps involved in one-voter-one-vote scenario:

*Step 1- Registration from Population List:*

We are provided with a population list of eligible persons to vote. Our system assumes the correctness of this supplied list. The registration process is handled under administrative control, but we have already ensured the authenticity of the administrator through the workflow above. During registration, there should be checking to avoid erroneous transaction, e.g. duplicate voter, non-unique user name-password combination etc. We have to simultaneously make sure that no eligible person is deprived of registration.

*Step 2- Vote Casting:*

Before casting his/her vote, every registered voter has to log into the system using the pre-specified user name-password combination for a particular election. We have to ensure the authenticity of the voter and also check against any attempt of

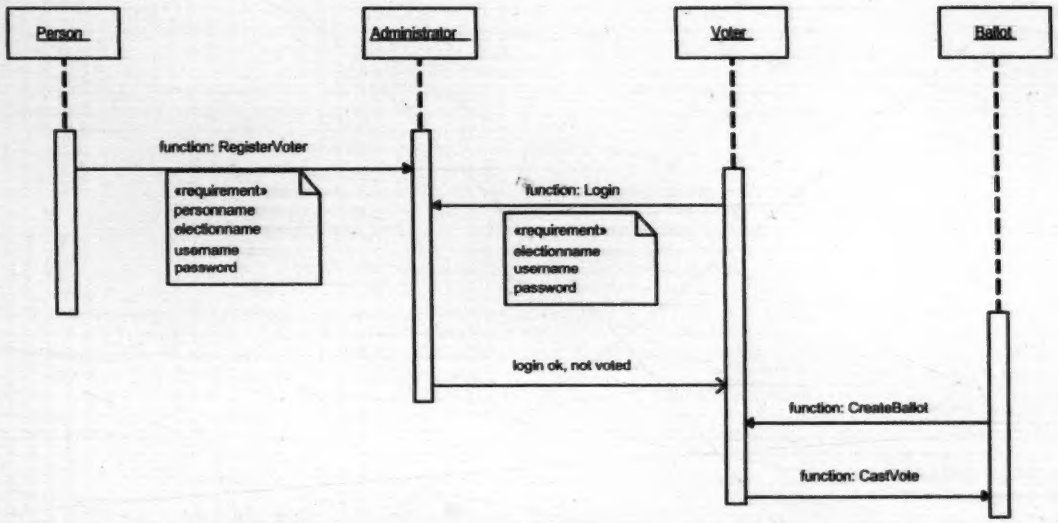


Figure 6.3: Sequence diagram of one-voter-one-vote scenario

multiple vote casting. The rules for voter login are similar to the admin login rules, except that we want to ensure that any redirect to the ballot page is guarded by both a valid user login, a check that they are eligible to vote, and they have not already voted.

In Figure 6.3 we present a high-level sequence diagram for the one-voter-one-vote scenario. In the next few paragraphs, there will be step-by-step evaluation of the different steps.

*One-Voter-One-Vote Scenario*

*Step 1- Voter Registration from Population List:*

Once logged on, the administrator has the option to register a person for a particular election. We select the person from an externally supplied list. The interface of Fig. 6.2 provides the opportunity to populate the *Election* and *Person* lists from the database. The following JSP code snippet is employed to do the caching of drop-down lists using an iterator that goes through every entry in the corresponding tables of the *Election* and *Person* classes in the database:

```
1  <form method=post action="add_voter.jsp">
2  <input type="HIDDEN" name="SUBMITTED" value="T">
```

```
3  Election:  <SELECT NAME="electionname">
4  <OPTION VALUE="NEW Election">Select One
5  <%
6  Election.loadAllElections();
7  Iterator it = Election.getElections();
8  while(it.hasNext()){
9  Election elt = (Election) it.next();
10  %>
11  <OPTION VALUE="<%= elt.getElectionName()%>" ><%= elt.getElectionID()%>:<%=
 elt.getElectionName() %>
12  <% } %>
13  </SELECT>
14  <p><p>
15  Person:  <SELECT NAME="personname">
16  <OPTION VALUE="NEW Voter">Select One
17  <%
18  Person.loadAllPersons();
19  Iterator it1 = Person.getPersons();
20  while(it1.hasNext()){
21  Person psn = (Person) it1.next();
22  %>
23  <OPTION VALUE="<%= psn.getName()%>" ><%= psn.getPersonID()%>:<%=
 psn.getName() %>
24  <% } %>
25  </SELECT>
```

This code sample involves quite a few steps, like: (a) we need to check the existence of a method ("loadAll" in this case) that loads every entry of a particular table, (b) we need to check the presence of an iterator function that goes through every loaded entry and (c) we need to select the appropriate value from the uploaded list.

In addition to selecting appropriate election and person names, we also have to provide a username and password during registration. These are going to be used by the voters during the vote-casting phase. When we try to post this information to the database we have to ensure the uniqueness of user-name-password data. We also have to ensure that the same person is not registered more than once for a particular election, as in Code 2.

In this portion of code, after checking that all the required data is correctly submitted, we go forward to create a new voter record in the database. While receiving inputs for voter registration there are several points where SCL can be applied to ensure that the intended scenarios are correctly represented within the code. First,

we have to make sure that our code receives inputs for all the required parameters from the user interface. In this case, new voter registration requires four different inputs from user: username, password, electionname, and personname. We have to ensure that (i) we receive the inputs after the SUBMIT button is pressed and (ii) we receive inputs for all the four parameters.

Code 2. Voter Registration Code

```
1  if (request.getParameter("SUBMITTED") != null) {
2  newvoter.setUserName("username");
3  newvoter.setPassword("password");
4  newvoter.setElectionName("electionname");
5  newvoter.setPersonName("personname");
6  try {
7  if (newvoter.validateVoter()) {
8  newvoter.createVoter();
9  response.sendRedirect("votercreate.jsp");
10  }
11  } catch (com.DuplicateUserNameException e) {
12  newvoter.addFieldError("username", ":This User Name already
 in use.  <BR>");
13  }
14  }
```

This is again typical of web forms. The code that processes the form must fetch each of the fields from the form and then set the corresponding attribute in the object being manipulated.

*Intent:* In SCL rule [SRC3] below, the first task is to define some simplifying variables. The *add_Voter* variable is defined as being the class *add_Voter_jsp* within the package *jsp*. This class, for the purpose of handling Http built-in functions, needs to extend the classes *HttpServletRequest* and *HttpServletResponse* of *javax.servlet.http* package. SCL considers these two classes as a parameter to our class *add_Voter* and we declare them using names *HttpServletRequest* and *HttpServletResponse* respectively. While we define the variable *jspServeiceVoter* we declare this variable as a method of the class *add_Voter* using *HttpServletRequest* and *HttpServletResponse* as parameters.

Once these variables are defined, we can state field-by-field rules for all the four parameters in question for the voter registration task.

According to our project design, when we check whether we have received the input for username we have to make sure that:

[i] username input is received after a call to the *getParameter("SUBMITTED")* function which represents the action of clicking the "SUBMIT" button (in other words this means that the existence of *setUserName()* function should be dependent upon the existence of *getParameter()* function)

[ii]we have called the *setUserName("username")* function (we need this because even if we do not call the *setUserName()* function at all there will not be an error message from SCL)

*Rule:* The SCL rule to express these two conditions looks like this:

*[SCR3] SCL Rule : Check all the required inputs are correctly accepted by code*

// Define some class variables to be used by SCL rules

```
1  add_Voter as class("jsp.add_Voter_jsp")
2  HttpServletRequest as class("javax.servlet.http.HttpServletRequest")
3  HttpServletResponse as class("javax.servlet.http.HttpServletResponse")
4  jspServiceVoter as method(add_Voter, "_jspService", HttpServletRequest,
   HttpServletResponse)
```

// part[i]: input for User Name comes under the control of getParameter() function

```
1  e:  exprs(jspServiceVoter) holds
2  ( (name(method(e))="setUserName")
3  =>
4  ex ce:  conds(e) holds ( name(method(ce))="getParameter" )
5  )
```

// part[ii]: this checks whether there is actually a call to the setUserName() function

```
1  exist e:  exprs(jspServiceVoter) holds
2  name(method(e))="setUserName"
3  error(<<add_Voter, jspServiceVoter>>, "This method should have
   called
4  setUserName().") // 1st of the four parameters is provided
```

*Evaluation:* Application of these two parts ([i] and [ii]) of [SCR3] ensures that we always call the *setUserName()* function under the control of *getParameter()* function. The same type of checking can be applied to the other three parameters: password, electionname and personname.

*[SCR3 contd.] SCL Rule contd.: Check all the required inputs are correctly accepted by code*

// input for Password

```
1e:  exprs(jspServiceVoter) holds   ( (name(method(e))="setPassword")
2  =>
3  ex ce:  conds(e) holds ( name(method(ce))="getParameter" )
4  ) // part[i]

1  exist e:  exprs(jspServiceVoter) holds
2  name(method(e))="setPassword"
3  error(<<add_Voter, jspServiceVoter>>, "This method should have
   called
4  setPassword().") // part[ii]
```

// input for Election Name

```
1e:  exprs(jspServiceVoter) holds   ( (name(method(e))="setElectionName")
2  =>
3  ex ce:  conds(e) holds ( name(method(ce))="getParameter" )
4  ) // part[i]

1  exist e:  exprs(jspServiceVoter) holds
2  name(method(e))="setElectionName"
3  error(<<add_Voter, jspServiceVoter>>, "This method should have
   called
4  setElectionName().") // part[ii]
```

// input for Person Name

```
1e:  exprs(jspServiceVoter) holds   ( (name(method(e))="setPersonName")
2  =>
3  ex ce:  conds(e) holds ( name(method(ce))="getParameter" )
4  ) // part[i]

1  exist e:  exprs(jspServiceVoter) holds
2  name(method(e))="setPersonName"
3  error(<<add_Voter, jspServiceVoter>>, "This method should have
   called
4  setPersonName().") // part[ii]
```

Inspecting these rules we find a general pattern where we have to make sure that a particular function (dependent) is called under the control of another function (independent). We can generalize rules like these in this way in SCL:

*Intent:* Generalized SCL rule.

*Rule: [SCR4] Generalization of SCL rule to check functional dependency*

```
1  CheckFunctionDependency(Class1, M_dependent, M_independent)
   as
2  (
```

```
3  e:  exprs(Class1) holds
4  ( name(method(e))=M_dependent
5  =>
6  exist ce:  conds(e) holds name(method(ce))=M_independent
7  )
8  &
9  (exist e:  exprs(Class1) holds
10  name(method(e))=M_dependent
11  )
12  )
```

Later, by providing necessary parameters we can use this generalized rule:
[SCR5]

```
1  p:  packages, c:  classes(p) holds
2  (
3  (c)="add_Voter"
4  =>
5  CheckFunctionDependency(c, "setUserName", "getParameter")
6  )
7  error(<<c>>, "Correct Parameter setting method is not called")
```

*Evaluation:* After all the necessary voter registration information is accepted we validate the supplied data before moving onto the database. This is achieved by calling the createVoter function under the control of the validateVoter function. We can utilize the parameterized general function declared in the previous paragraph by providing the corresponding function name:

*Intent:* Use the generalized SCL rule for specific conditions.
*Rule:* [SCR6]

```
1  p:  packages, c:  classes(p) holds
2  (
3  (c)="add_Voter"
4  =>
5  CheckFunctionDependency(c, "createVoter", "validateVoter")
6  )
7  error(<<c>>, "Correct Parameter setting method is not called")
```

*Evaluation:* Having achieved all the preconditions of receiving inputs from the user interface, we try to create a new voter record in the database. In this stage our primary intention is to make sure that no voter is registered more than once. We query the database with the supplied voter information before posting it to the record. The database is queried with the supplied voter information to check

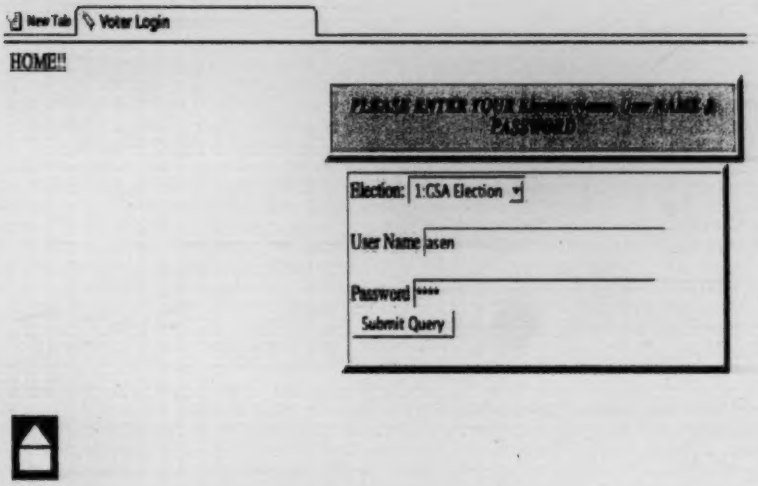


Figure 6.4: Voter Login

whether any existing record matches with the new one. If not, we proceed to store the supplied information in the database. Our parameterized function can be applied again at this stage to make sure that the findVoter function is called from within the createVoter function.

Assuming that every checking stage is successful a new voter record is created for a particular election. When the vote casting phase of that election is initiated, registered voters log into the system using their pre-recorded information to cast vote. This leads us to the Step 2 of the one-voter-one-vote scenario.

*One-Voter-One-Vote Scenario*

*Step 2- Vote Casting:*

When the vote casting is started, every eligible voter has to log into the system with their pre-recorded username-password combination for a particular election. We used the interface of Fig. 6.4 for voter login. Here, to populate the list of *Election* we applied the same principle used during Step 1 to populate *Election* and *Person* lists. When the required information for voter login is submitted, we query the database to ensure the existence of the voter record. At the same time we check whether a voter has already voted for a particular election.

This voter authentication phase [Code 3] is similar to the previous examples because we move towards querying the database after we receive inputs from the user which, in this case, is the voter trying to cast their vote. Here, the *findVoter* function authenticates the supplied voter information and the *checkVoted* function checks whether this particular voter cast his/her vote already. We can apply the previous parameterized function by considering that both the *findVoter* and the checkVoted functions are dependent upon the existence of the getParameter function. Upon success of these checks, the voter is provided with the ballot page for casting their vote.

Code 3. Voter Authentication

```
1  if(request.getParameter("SUBMITTED") != null) {
2  name = request.getParameter("username");
3  if(findVoter(request.getParameter("username"), request.getParameter("password"),
 request.getParameter("electionname")) == null)
4  {
5  error = "No such Voter found, try again..";
6  }
7  else {
8  if(checkVoted(request.getParameter("username"), request.getParameter("password"))
9  {
10  out.write("<p>WELCOME<p> To CAST VOTE : <a href="voterlogin.jsp?usernamename=");
11  out.print( name );
12  out.write(>CLICK HERE</a><br>);
13  return;
14  }
15  else
16  {
17  out.write("<p>SORRY, You Have Already Voted...<p> To TRY AGAIN
 : <a href="voterverify.jsp">CLICK HERE</a><br>");
18  return; } } }
```

# Chapter 7

# Qualitative Experience

As we worked with both Java code and SCL rules for our project we observed the following.

While we were working on our project, there are several issues that we found difficult or impossible to represent in SCL. If we revisit the example of calling a function with parameters [Code 1 and SCR2], it is evident that we cannot check the actual parameters supplied through the Java function call. We can only check whether or not the function sendRedirect is dependent on the function findAdmin. But there is no way with SCL to check whether the findAdmin function is called with the right parameter. If we extend this further, SCL is also unable to check the effect of any data supplied by a function call on the underlying database. For example when we apply SCL rules for voter registration we can only check whether the appropriate function calls are made, but we cannot check the effect of the supplied parameter on the underlying database. This is a significant limitation of SCL.

There is another issue connected with the dependency checking that might be of interest. When we try to check functional dependency, we have to ensure that the dependent function is under the control of the independent one via any of if, for, while or such other controller. We cannot check if a particular function call comes after another one if the former one is not conditionally dependent on the later one. This limitation of SCL actually presents us with a kind of standard SCL pattern where we need to use the SCL conditional dependency operators for a block in our program enclosed by these controllers. Or in other words, using SCL conditional dependency operators somewhere in the testing code means that portion of SCL

code corresponds to a block of source code enclosed in some type of controller.

This inability of SCL to check functional ordering applies to any part of the source code. SCL will give us the information about whether a particular function is present within a block of code, but there is no mechanism in SCL to verify the exact moment of its execution within the whole block. This might cause some implications in SCL checking when two functions (say, f1 and f2) are not conditionally dependent but there should be a certain order of their execution. Say, we want to verify the presence of both the functions in a block of code. At the same time, there is a requirement that f2 is executed after f1. We can apply SCL only to test their presence, not their order of presence in the code.

Because they are structural, the creation of SCL rules and the writing of Java code occur simultaneously. We found that attempting to create the SCL rule tended to influence our design. We are going to present an interesting scenario where the inability of SCL to check the order of execution of functions actually prompted us to change the design of code. Let us investigate the vote casting scenario where the precondition is that the person attempting to cast vote must not have done so before. After this checking is done we need to store his/her vote information in the database and then mark him/her as already voted to prevent any duplicate vote. One possible vote casting scenario might look like this:

```
1 public class Voter {
2 public void CastVote(voterID, voteData) {
3 if (checkVoted(voterID)) return;
4 saveVote(voteData);
5 setVoted(voterID);
6 otherFunction1();
7 otherFunction2();
8 }
9 }
```

Here, we can apply SCL only to check the presence of the two functions saveVote and setVoted, not to check their order of execution. But to replicate the actual vote casting steps, we should ideally execute them in order without any other function like otherFunction1 or otherFunction2 being executed in between them. In other word, the two vote related functions within the CastVote function should be atomic.

But SCL has no mechanism to guarantee the atomic execution. This forces us to modify the code a little bit where we replace the two concerned functions with a new function which actually calls them separately. Then SCL will be able to test the presence of the new function. Our modified code might look like this:

```
1  public class Voter {
2  private void commitVote(voterID, voteData) {
3  saveVote(voteData);
4  setVoted(voterID);
5  }
6  private void CastVote(voterID, voteData) {
7  if (checkVoted(voterID)) return;
8  commitVote(voterID, voteData);
9  otherFunction1();
10  otherFunction2();
11  }
12  }
```

SCL can check the presence of the commitVote function which has only the two required functions within it. So, by checking the existence of commitVote function through SCL, we are testing the atomic behavior of the two other functions. That is, SCL forced us to write a function that encapsulates the atomic operation - a good design rule in any system. There are many similar design rules, such as one exit from a loop, that use a convention on program structure to help the reader in their understanding of program behavior.

This example illustrates the fact that thinking about a SCL rule forces you to think about what the structure says about the behavior of the code, since intentions are ultimately behavioral. This has much the same effect as a requirements review and leads to earlier detection of specification errors.

# Chapter 8

# Comparison with other testing tool

One of the most potent disadvantages of SCL is that it is basically a static testing tool. In this project of ours we do a lot of database access, which means the state of the database is bound to change during runtime. This leaves us with no option but to use other testing techniques that are dynamic in the sense that they are able to test the real-time change of database. Here comes the applicability of unit testing. In the unit testing methodology we have to ensure that all the functions or procedures within a given class execute properly and produce the correct output (if any) under every possible set of input parameters. To compare the unit testing rules with the SCL rules described earlier let us consider how we are going to apply unit testing for the new voter registration function. In our project we wrote our own unit testing codes which simulate the behavior of the standard JUnit testing tool. Writing our own test code gave us the flexibility to execute and test the behavior of various parts of the system according to our design flow. In fact, within an specific test suite that we wrote we called several small functions to perform a sequence of operations; which gave us the flavor of a controlled integration test of the system. For example in the following test script of voter registration, at first we executed the functions to set different parameters, followed by the validation of the parameters, creation of the record in the database and then we tried to create duplicate record.

Unit Testing Code : Voter Registration

```
1  Voter v = new Voter();
2  v.setUserName("user1");
3  v.setPassword("password1");
```

```
4  v.setElectionName("GSA Election");
5  v.setPersonName("person1");
6  if (v.validateVoter()) System.out.println("Input Validated");
7  try {
8  v.createVoter();
9  System.out.println("Test Pass:  Create Voter");
10  }
11  catch (Exception e) {
12  System.out.println("Test Fail:  Could Not Create Voter");
13  }
14  try {
15  v.createVoter();
16  System.out.println("Test Fail:  Could Create Duplicate Voter");
17  }
18  catch (DuplicateUserNameException e) {
19  System.out.println("Test Pass:  Exception Thrown and Caught
 When Create Duplicate Voter");
20  }
```

Output produced by this testing code when the java code is run by ant build tool:

```
1  adtest:
2  [java] Turbine:  init() Ready to Rumble!
3  [java] Input Validated
4  [java] Test Pass:  Create Voter
5  [java] Voter is:  com.polling.admin.Voter@ac4d3b
6  [java] Voter User Name is:  user1
7  [java] Voter Password is:  password1
8  [java] Test Pass:  Exception Thrown and Caught When Create
 Duplicate Voter
```

The obvious difference between this testing rule and the SCL rule shown before [SCR3] is that we can provide real input to the database for testing with unit test. This enables us to inspect the run-time behavior of our source code when the state of the database is changed dynamically. This fact is significant in the sense that it clearly shows the inability of SCL to be effective for dynamic testing.

Having said that, we can apply SCL side-by-side unit testing to identify potential tests. In the above example, before writing unit testing, we can apply some SCL rules that will identify which functions to unit test. By applying SCL, we can make sure that all the parameter setting functions (like setUserName, setPassword, setElectionName, setPersonName) are executed correctly. Then we can apply unit testing rules to check the effect of the supplied parameters to the underlying database. SCL alone cannot check the effect of parameters on the system. But once

we check the flow of operation through SCL, we can apply small unit tests within this flow to validate the effect of user-supplied data on the whole system.

This ability of SCL to uncover the flow of execution makes SCL a good aid for white box testing. We know white box testing normally tests paths within a unit or block of code. But it can also test paths between units during integration test. So SCL will give us the test paths needed for white box testing by checking the flow of operation within the system.

But even this cooperation between SCL and other testing methods is not sufficient in some cases. If, by any chance, the voter registration steps are executed more than once in the source code, the previous idea of applying SCL along with other tool might not be good enough to detect the duplicate occurrence of the same step. To avoid a scenario like this we may need to employ manual code inspection.

SCL can also help us detect the inadequacy of some tests dependent on specific preconditions. For example, if we test an execution flow which requires the fulfillment of an assumption like two different classes within the system cannot access each other and that assumption is somehow broken in the code, SCL can help us detect the illegal class access. Finding these types of violations within the source code can be done by SCL; which will enable us to re-design the existing test code.

We can confidently deduce that SCL, at its current state, is strong enough to check the existence of pre-specified user intents in the source code. But when it comes to evaluating the dynamic behavior of a bunch of code we cannot rely on SCL; rather we have to use different testing techniques, like the manual unit testing codes as discussed in the previous paragraph or an standard testing tool like JUnit. Having said that, our experience with SCL gives us the confidence that it can help us to design test scripts suitable for JMeter testing by checking that certain testing assumptions still hold. JMeter is a widely used testing tool which can help us perform integration, stress, and load test. But to apply JMeter for these types of tests we need to come up with some testing scripts suitable for JMeter that will simulate the behavior of a sequence of actions performed over the system. SCL might tell us how many and which cases should be tested within a block of code which will give us a clear strategy while creating test scripts for JMeter.

# Chapter 9

# Conclusion

Capturing intentions through the static analysis of source code is just one more tool to address security. Application behavior is yet another aspect of security issues. For example, incorrect exception handling can expose a system to attacks. These kinds of analyses require dynamic analysis tools, such as those developed by Li, Hoover, and Rudnicki [10]. Just as moving from paper to electronic ballots creates an enormous increase in implementation complexity, we should expect our tools for analysing security issues to do the same. Our experimentations with web-based voting system give us the confidence that an static source code analysis tool like SCL is useful enough to capture pre-specified user intents in the source code. At the same time we should apply other testing tools or approaches like unit testing to judge the dynamic behavior of the code. Due to the limitations of SCL expressiveness concerning flow of execution and the strict dependency of SCL rules on the design of the system, we recommend an approach where we start with a well-planned design of the system to correctly express programmer intents. Then the programmer should concentrate on developing his system alongside the SCL testing code, while conforming to the design strictly. This will enable the programmer to come up with the system that is following both the design constraints and the SCL regulations. Any modifications carried on the design because of any change in user intents should be reflected by the simultaneous alterations of the source code and the SCL rules corresponding to that code.

We have developed the idea that SCL, at its current state, is not good enough for a real-time analysis of the program; its applicability is limited to the static case

which enables us to verify the program structure and to enforce some complex properties. With the introduction of dynamic testing capabilities, SCL can be applied as a tool good enough for both static and dynamic analysis of program.

# Chapter 10

# Acknowledgments

We are grateful to Daqing Hou (dhou@clarkson.edu) for access to and help with SCL. This research was supported by a Natural Sciences and Engineering Research Council of Canada Discovery grant.

# Bibliography


[1] Beer, I., Eisner, C.: *The Temporal Logic Sugar.* 13th International Conference on Computer Aided Verification, LNCS, Vol. 2102, pp. 363-367.

[2] Ernst, M.: *Ststic and Dynamic Analysis: synergy and duality.* Workshop on Dynamic Analysis, May 9, 2003.

[3] Herrnson, P., Niemi, R., Hanmer, M., Bederson, B., Conrad, F., Traugott, M.: *The Importance of Usability Testing of Voting Systems.* Electronic Voting Technology Workshop, August 1, 2006.

[4] Hou, D., Hoover, H.J.: *Using SCL to Specify and Check Design Intent in Source Code*, IEEE Transactions on Software Engineering, Vol 32, Number 6, June 2006, pp 404-423.

[5] Keller, A., Mertz, D., Hall, J., Urken, A.: *Privacy Issues in an Electronic Voting Machine.* ACM workshop on Privacy in the Electronic Society, 2004.

[6] Kicimen, E., Wang, H.: *Live Monitoring: Using Adaptive Instrumentation and Analysis to Debug and Maintain Web Applications.* 11th Workshop on Hot Topics in Operating Systems, San Diego, May 7-9, 2007.

[7] Kohno, T., Stubblefield, A., Rubin, A., Wallach, D.: *Analysis of an Electronic Voting System.* IEEE Symposium on Security and Privacy, 2004.

[8] Laskowski, S., Redish, J.: *Making Ballot Language Understandable to Voters.* Electronic Voting Technology Workshop, August 1, 2006.

[9] Lauer, T.: *The Risk of e-Voting.* Electronic Journal of e-Government, 2004.

[10] Li, X., Hoover, H.J., Rudnicki, P. *Towards Automatic Exception Safety Verification*, Proceedings of the 14th International Symposium on Formal Methods. Aug 21–27, 2006, Hamilton, Ontario, Canada. pp 396–411. Springer, LNCS 4085.

[11] Mercuri, R.: *Generic Security Assessment Questions.* www.notablesoftware.com.

[12] Neumann, P.: *Security Criteria for Electronic Voting.* 16th National Computer Security Conference, Maryland, September 1993.

[13] Rubin, A.: *Security Considerations for Remote Electronic Voting over the Internet.* Communications of the ACM, Vol 45, Issue 12, December 2002.

[14] Sampio, A., Vasconcelos, A., Sampio, P.: *Towards Reconciling Quality and Agility in Web Application Development.* International Workshop on Web Quality, Munich, July 27, 2004.

[15] *California Voting Systems Review.* www.sos.ca.gov/elections/elections_vsr.htm.